# Exercise generation on language specification

J.João Almeida[1], Eliana Grande[2], and Georgi Smirnov[2]

Universidade do Minho, Braga, Portugal
[1] Department of Informatics, Algoritmi
[2] Department of Math and Applications

**Abstract.** Exercise generation on language specification is a challenging problem, because of the richness of the objects in the domain. In this paper we discuss MGBEG (Meta-Grammar-Based Exercise Generator) – a toolkit for exercise generation on context-free languages. MGBEG approach is based on a meta-grammar formalism and tool, used to define a set of similar exercises. MGBEG is simple attributed grammar used to describe the set of valid exercise (and randomly generate one of them). Each exercise typically contains several attributes calculated during the generation steps: namely, one or more formal specification of the language (context free grammar); the exercise statement; other information such as examples, common mistakes, validation data, to be used in the construction of the exercise statement, solution, and assessment steps. Complementary the toolkit provides a grammar module, with functionality for grammar comparison, sentence generation and recognition; a template engine (to help in textual attributes calculation).

## 1 Introduction

It is well known that a solid knowledge of language specification formalisms and tools is very important in computer science [1]. Computer science curricula always include modules in these sensible subjects and we can find several good exercises on this topic [1].

When we deal with automatic generation of language specification exercises (capable of producing sets of similar exercises), we need to process the core language specification artifacts and to generate, transform, compose them to produce the necessary exercise statements, exercise solutions, validation/assessment functions. This is not easy and clearly needs guidelines and support tools.

In this section, we describe the context of this work. Then we discuss typical artifacts of a language specification exercise. Finally, we present MGBEG and demonstrate its capability of generating exercise.

### Context

This work concerns exercise generation tools. The main issues addressed here are the study of exercise structure, and understanding and description of exercise generation processes.

[1] The same knowledge is important for applied linguistics and other related areas.

In fact we claim that exercise generation tools should provide (whenever possible) a rich exercise structure, including elegant exercise statements (like in the case of math exercises [2]), exercise solutions, answer assessment functions, information about the student's success and achievements.

### Language exercises artifacts

A typical exercise on introductory language specification, deals with:

- providing a natural language description;
- providing a language specification (regular expression, context free language, automata);
- asking for examples of valid sentences;
- asking if some sentences belong to the language;
- conversion between language specification formats;
- asking for derivation tree of a sentence;
- sentences semantics (this topic is not covered in this paper).

Clearly the sketched exercises involve the manipulation of a set of language artifacts that are not independent, and cannot be generated separately.

### Mgbeg approach

The main result of this work is the development of MGBEG exercise generator allowing one to generate a large amount of exercises from a grammar-based scheme.

The process of language specification exercise generation can be seen as a:

- definition of a set of similar grammars – this definition is done using a meta-grammar;
- complementation of the meta-grammar with a set of attribute rules and templates capable of calculating alternative forms of the grammar and objects derived from the grammar (Example of these attributes are: a natural language description of the exercise, examples of valid sentences, distractors, and data structures to help to assess the student's answers);
- use of a top down random generator to choose a grammar and associated attributes;
- use of this information to fill the exercise templates.

A primitive approach to exercise generation consists in building a exercise database, and can schematically be represented by the following:

```
type exercise-db: (exer_stat, result)*
func generate(e):
   x ← choice(e)
   return(x)
```

A more sophisticated approach is based on general exercise schemes and can be described as follows:

```
type exercise-db: exer-sch*
     exer-sch: (template, table)
     table: (vars, result)*
func generate(e):
   x  ←  choice(e)
   row ← choice(x.table)
   exer-stat ← x.template(row.vars)
   return (exer-stat, row.result)
```

This approach helps to better understand the structure of the exercise and to generate a large number of exercises.

MGBEG, presented in this paper, generalizes the above approach to multi-level templates, variables and results. The generalization of the exercise schemes is syntactically described in terms of a meta-grammar; in a simplified way, the random choices correspond to generated metagrammar sentences.

The paper is organized as follows. In section 2 we discuss the concept of meta-grammar, its structure and attributes. In section 3 we briefly describe the MGBEG grammar module, used in the basic meta-grammar engine and in the attribute rules. In section 4 we discuss some details about assessment of student's answer. Finally we present some conclusions.

## 2 Meta-grammar

As stated before, a meta-grammar (MG) is a grammar used to describe and generate grammars. The set of all possible derivation chains, defines the group of exercises covered by the MG [9]. Note that here the term "meta-grammar" is not referring to the grammars Meta-S (§-grammars), connected to the adaptive grammars [8].

### 2.1 Meta-grammar structure

A meta-grammar is a tuple $MG = (MN, MT, MP, MS)$ where:

- $MN$ is a non-empty set of non-terminal symbols or variables;
- $MT$ is is the set of terminal symbols or vocabulary;
- $MP$ is set of productions, where each production is of the form $(lhs, rhs, atr)$, where $lhs \in MN$, $rhs \in (MT \cup MN)^*$ and $atr$ is a set of attribute rules;
- $MS \in MN$ is a start symbol or axiom.

In order to generate exercises, the meta-grammar, in addition to generation of a grammar, also generates attributive information guided by the attribute rules defined in each production.

**Exemplo:**

[MS] → S : [K] ;
{Consider the language of $K. Create a grammar that generates this language.} .
[K] → E [Sep] S | E ; E : [T]
{the lists of $T separated by a $Sep}
‖ ( L ) ; L : L FL | $\varepsilon$ ; FL : S | [T]
{generalized list of $T separated by parentheses } .
[Sep] → , {comma}
‖ / {bar}
‖ - {dash} .
[T] → INT { integers (INT)}
‖ ALPHA {letters (ALPHA)} .

**Table 1.** Example of Meta-Grammar list or generalized list of integers or letters.

### 2.2 Example

In this example (see Tab.1) we define a MG to generate exercises of lists and generalized lists of integers or letters.

In this meta-grammar we have $MN = \{MS,\ K,\ Sep,\ T\}$ (the set of non-terminal symbols), the axiom is $MS$, $MT = \{S,\ E,\ ',',\ L,\ FL,\ ALPHA,\ INT,\ :,\ ;,\ -,\ |\}$ (the set of terminal symbols). In order to have clearer reading, in meta-grammars, we write non-terminals inside boxes. Each sentence generated by the meta-grammar is a grammar and its associated attributes (in this case one single attribute – the *exercise statement* for the grammar). In Tab. 2 we present a derivation chain and a syntactic tree for the sentence "S : E , S | E ; E : INT ;" (in fact a grammar for language of the lists of integers (INT) separated by commas).

Now we will create a learning activity using the above meta-grammar. This activity is composed of two steps:

1. Using the meta-grammar, we create two grammars (field "g") and associated attributes (in this case, the only attribute is the exercise statement, field "e")
2. Using these *language artifacts* we fill the gaps of the activity template.

Consider the following activity definition code :

```
a = mgbeg("lists.mg", exercise=2, example=3)
                  // a: set of artifacts (2 grammars, 3 examples each)
#question
    #a[0].e                      // exercise statement of first grammar
    Examples of valid phrases:
    . #a[0].example[0]           // example phrases
    . #a[0].example[1]
```

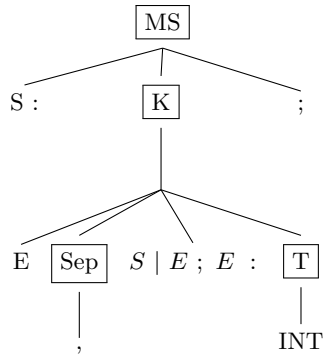
| Derivation chain | Derivation tree |
|---|---|
| $\boxed{\text{MS}} \Rightarrow S : \boxed{\text{K}} ;$<br>$\Rightarrow S : E \boxed{\text{Sep}} S \mid E ; E : \boxed{\text{T}} ;$<br>$\Rightarrow S : E , S \mid E ; E : \boxed{\text{T}} ;$<br>$\Rightarrow S : E , S \mid E ; E : INT ;$ |  |

**Table 2.** The derivation tree of the sequence "S : E , S | E ; E : INT ;".

```
    . #a[0].example[2]
#result
    #a[0].g                        // the created grammar
#validation
    gramequiv(#1, #a[0].g)         // submission grammar = first grammar
#question
   (a) #a[1].e
   (b) write a valid sentence.
#result
    (a) #a[1].g
    (b) Example of valid phrases:
      . #a[1].example[0]
      . #a[1].example[1]
      . #a[1].example[2]
#validation
    (a)gramequiv(#1, #a[1].g);   // submission grammar=second grammar
    (b)validsent(#a[1].g, #1)    // is a valid sentence
```

After processing the previous, several document are created, namely:

**(i) The exercise statement**

1. Consider the language of the lists of letters (ALPHA) separated by a dash. Create a grammar that generates this language.
   Examples of valid phrases:
   v - a
   d - o
   h - k - g
2. (a) Consider the language of the generalized lists of letters (ALPHA) separated by parentheses . Create a grammar that generates this language.

(b) Write a valid sentence.

**(ii) The solution**

1. S : E - S | E ;
   E : ALPHA ;
2. (a) S : ( L ) ;
   L : L FL | $\varepsilon$ ;
   FL : S | ALPHA ;
   (b) Example of valid phrases:
   ( )
   ( ( a ) ( ( b ( d ) ) ( ) ) )
   ( ( ( ) b e ) ( b a f ) x ( ) )

**(iii) The verification script** (not included)

### 2.3 Meta-grammar attributes

The construction of meta-exercises (models to generate exercises) is a complex process. The fragmentation of the exercises in parts to be combined later, generating elegant exercise statements are challenging problems.

To generate grammars with associated attributes (language artifacts), MGBEG metagrammars productions may use additional attribute rules. The following attribute rule formats are contemplated:

- `id:{...template with variables}` – to set an attribute *id* with the value of the template after expanding the variables;
- `id:!{...expression with variables}` – to set the value of attribute *id* with the result of calculating the expression after variable expansion;
- `{template with variables}` and `!{...expression with vars.}` – (identical to the previous) – to set the default attribute *e* (exercise statement); this type of rules were used in the previous example.

In the attribute rules (templates and expressions) the variables refer to attributes of symbols present in the rhs of the production and are expanded with their values. The following formats are supported:

- `$A.id` – attribute *id* of the non-terminal symbols *A*
- `$A.g` – attribute *g* (the grammar) of the non-terminal symbols *A*
- `$A` – attribute *e* (the exercise statement) of the non-terminal symbols *A*

### 2.4 Example with complex attributes

Let us consider the following grammar:

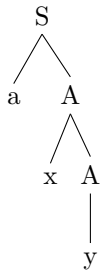
```
S : a A | b
A : x A | y
```

A common exercise is: *Write a valid sentence with at least 3 symbols, and present the correspondent syntax tree.*

In order to draw syntax trees, several tools (such as rsyntaxtree [7], phpsyntaxtree [6], or qtree) use a square-bracket tree notation:

```
[S a [A x [A y ] ] ]
```

for the tree




The following example shows how, using MGBEG, it is possible to construct, together with the above grammar, a correspondent regular expression definition, and a tree drawing grammar.

```
[G] →   S : a A | b ; [P]
        re:{ a $P.re + b   }
       stg:{ S1 : [ S a A1 ] | [ S b] ; $P.stg }

     ‖  S : A a | b ; [P]
        re:{ $P.re a + b   }
       stg:{ S1 : [ S A1 a ] | [ S b] ; $P.stg } .

[P] →   A : x A | y
        re:{ x* y  }
       stg:{ A1 : [ A x A1 ] | [ A y ] }

     ‖  A : A x | y
        re:{ y x* }
       stg:{ A1 : [ A A1 x ] | [ A y ] }

     ‖  A : x A | x y
        re:{ x* x y }
       stg:{ A1 : [ A x A1 ] | [ A x y ] } .
```

As a result of generation based on this metagrammar we have 6 tuples of the form:

```
g  : S → A a | b ; A → x A | x y      // Grammar
re : x*xya + b                        // Regular expression
stg: S1→[S A1 a] | [S b]; A1→[A x A1] | [A x y]  // Tree drawing grammar
```

This example illustrates the use of complex attributes to calculate elements for the exercise statement and assessment. In the assessment process we check if the submitted syntax tree belongs to the language generated by constructed tree drawing grammar.

## 3 Mgbeg grammar module

It was clear from the beginning of the project that it was crucial to have a rich toolkit with functions capable of manipulating grammars and other language artifacts.

Bellow we present some of the most used functions of the MGBEG module.

**gramequiv: grammar, grammar → bool** — given two grammars calculate if they describe the same language (several other grammar equivalence functions are possible [4]). This function is briefly described in section 4.

**validsent: grammar, sentence → bool** — check if sentence is valid.

**grampp: grammar → LaTeX** — grammar pretty printer in LaTeX

**bisongram: grammar → bison-parser** — generate and compile a simple Bison [5] parser for the grammar.

**gramgen: grammar → sentence** — randomly generates examples of valid sentences.

**mgramgen: metagrammar → (id → value)*** — top-down generator. Returns a grammar, exercise statements and related attribute information; described in Alg. mgramgen.

**mgbed: metagrammar, options → seq of (id → value)*** — generate one or more grammars, attributes and examples.

**Function** mgramgen(mg,ax): Generate a grammar and exercise statement

```
Input: mg: meta-grammar
Input: ax: axiom ∈ NT
Output: r: mapping id → (grammar, statement or language artifact)
begin
    if ax ∈ T then                                  //grammar text is just "ax"
        return (g → ax)
    if ax ∈ NT then
        r[g] ← "                                    // initialize grammar
        rhss ← mg[ax]
        (rhs,attrules) ← choice(rhss)               //randomly choice a production
        for s ∈ rhs do
            aux ← gen(mg,s)
            r[s] ← aux
            r['g'] ← r['g']+aux['g']
        for (id : t) ∈ attrules do                  //if rule has no id, def: e=exer. statem
            if t is !{...} then                     //Perl expression template
                aux ← template-expand-vars(t,r)
                aux ← eval(aux)
            if t is {...} then                      //a normal template
                aux ← template-expand-vars(t,r)
            r[id] ← aux
    return (r)
```

## 4 Assessment

In [3], we propose a new assessment method. It consists of the following steps:

1. Transformation of a context-free grammar into a system of formal nonlinear equations [10, 11];
2. Substitution of the terminal alphabet letters by randomly generated $n \times n$-matrices [2];
3. Numerical solution of the system of nonlinear matrix equations;
4. Doing again the previous steps $N$ times.

Numerical solution of the system of nonlinear matrix equations generated by a grammar is equivalent to the computation of the sum of the matrix series obtained by substitution of the terminal alphabet letters by matrices. If the sums of two formal power series corresponding to two grammars coincide for all possible substitutions of $n \times n$-matrices, then the series are identical for a very large class of context-free grammars. This observation allows us to use a probabilistic approach to compare two grammars. Namely, if the solutions of the respective systems of matrix equations coincide in $N$ successive random matrix substitutions, we conclude that the grammars generate the same language.

This assessment method is illustrated in the following:

| Grammar $G_1$; axiom = S | Grammar $G_2$; axiom = B |
|---|---|
| S $\rightarrow$ ( L )<br>;<br>L $\rightarrow$ L FL<br>$\mid \varepsilon$<br>;<br>FL $\rightarrow$ S<br>$\mid$ ALPHA<br>; | B $\rightarrow$ ( K )<br>;<br>K $\rightarrow$ ALPHA K<br>$\mid$ B K<br>$\mid \varepsilon$<br>; |
| $\downarrow$ After transformation to mathematical analysis domain | |
| System of equations for $G_1$ | System of equations for $G_2$ |
| S = ( L ),<br>L = L FL + I,<br>FL = S + ALPHA. | B = ( K ),<br>K = ALPHA K + B K + I. |
| $\downarrow$ Values of the axioms after solving the systems of equations | |
| $\mathbf{G_1.S} = \begin{pmatrix} 0.010945941 & 0.011964876 \\ 0.004055173 & 0.008135430 \end{pmatrix}$ | $\mathbf{G_2.B} = \begin{pmatrix} 0.010945941 & 0.011964876 \\ 0.004055173 & 0.008135430 \end{pmatrix}$ |
| $\downarrow$ | |
| Conclusion: $G_1$ and $G_2$ are equivalent | |

[2] In general practical situations $2 \times 2$-matrices proved to be sufficient.

## 5 Conclusion

In this paper we presented an algorithm for exercise generation in the domain of formal languages, namely context-free grammars. The algorithm makes use of meta-grammars. The main advantages of this approach are:

- The use of meta-grammars helps us to structurize the exercise;
- The attributes organize the specification in a high-level declarative way;
- The use of metagrammars allows one to treat in a unified manner all the elements of exercise, such us the statement, examples, grammars, regular expressions, syntactic trees, assessment components.

The MGBEG system described here shows positive results as a tool for the learning process.

**Acknowledgments** The work of Eliana Grande was supported by PIQ IF Goiano through the fellowship no. 02/2013. The work of J.J. Almeida was supported by COMPETE: POCI-01-0145-FEDER-007043 and FCT – Fundação para a Ciência e Tecnologia within the Project Scope: UID/CEC/00319/2013.

## References

1. Aho, A.V., Sethi, R., Ullman, J.D.: Compilers: Principles, Techniques, and Tools. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA (1986)
2. Almeida, J., Araújo, I., Brito, I., Carvalho, N., Machado, G.J., Pereira, R.M., Smirnov, G.: Math exercise generation and smart assessment. In: Workshop of TICAMES (Information and Communication Technology in Higher Education: Learning Mathematics), CISTI-2013. pp. 1014–1019 (2013)
3. Almeida, J.J., Grande, E., Smirnov, G.: Context-Free Grammars: Exercise Generation and Probabilistic Assessment. In: 5th Symposium on Languages, Applications and Technologies (SLATE'16). OpenAccess Series in Informatics (OASIcs), vol. 51, pp. 1–8. Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik (2016)
4. Cousot, P., Cousot, R.: Grammar analysis, and parsing by abstract interpretation. In: Program Analysis and Compilation, Theory and Practice: Essays dedicated to Reinhard Wilhelm. p. 178–203 (2006), lNCS 4444
5. Donnelly, C., Stallman, R.: Bison. the yacc-compatible parser generator. Tech. rep., GNU (2004)
6. Eisenbach, A.: phpSyntaxTree - drawing syntax trees made easy. Tool page (2015), `http://ironcreek.net/phpsyntaxtree/`
7. Hasebe, Y.: RSyntaxTree - yet another syntax tree generator made with ruby and rmagick. Tool page (2016), `http://yohasebe.com/rsyntaxtree/`
8. Jackson, Q.T.: Disambiguation as a quantifiable computational process. Tech. Rep. 3 (2000), `https://www.researchgate.net/publication/2401401_Disambiguation_as_a_Quantifiable_Computational_Process`
9. José Neto, J.: Introdução à compilação. Livros Técnicos e Científicos, Rio de Janeiro (2016)
10. Salomaa, A.: Formal Languages. Acad. Press (1973)
11. Salomaa, A., Soittola, M.: Automata-theoretic aspects of formal power series. Springer (1978)